-------------------------------------------------

Selection Sort with Heuristic: Line-by-Line Explanation

-------------------------------------------------

-------------------------------------------------

[1–2] def selectionSort(arr):

def heuristic(arr):

- Defines the main sorting function `selectionSort`.

- Inside it, defines a helper function `heuristic` to measure how far elements are from their sorted positions.

-------------------------------------------------

[3–5] sorted\_arr = sorted(arr)

heuristic\_values = [abs(arr.index(x) - sorted\_arr.index(x)) for x in arr]

return heuristic\_values

- Creates a sorted copy of the input array.

- For each element, calculates how far its current index is from its correct (sorted) index.

- Returns a list of these heuristic (distance) values.

-------------------------------------------------

[7] print("Heuristic is used to calculate the distance of each element from its target sorted position.")

- Prints a message explaining the purpose of the heuristic.

-------------------------------------------------

[9–16] for i in range(len(arr)):

min\_index = i

for j in range(i + 1, len(arr)):

if arr[j] < arr[min\_index]:

min\_index = j

arr[i], arr[min\_index] = arr[min\_index], arr[i]

print(f"Step {i + 1}: Array: {arr} | Heuristic: {heuristic(arr)}")

- Runs the outer loop for each position `i` in the array.

- Finds the index of the smallest element (`min\_index`) in the unsorted part.

- Swaps the current element `arr[i]` with the smallest found element.

- Prints the array state after each swap along with the heuristic list.

-------------------------------------------------

[18] return arr

- Returns the fully sorted array after the selection sort completes.

-------------------------------------------------

[20] arr = list(map(int, input("Enter the array elements separated by space: ").split()))

- Takes input from the user as space-separated integers and converts them into a list.

-------------------------------------------------

[21] sorted\_array = selectionSort(arr)

- Calls the `selectionSort` function to sort the user-provided array.

-------------------------------------------------

[22] print("Sorted Array:", sorted\_array)

- Prints the final sorted array.

-------------------------------------------------

Possible Viva Questions

-------------------------------------------------

1️⃣ What is Selection Sort?

- A simple comparison-based sorting algorithm that repeatedly selects the smallest element and moves it to its correct position.

2️⃣ What is the time complexity of Selection Sort?

- O(n²), because of the nested loops.

3️⃣ What heuristic is used in this program?

- The absolute distance of each element from its target sorted index.

4️⃣ Why use a heuristic in sorting?

- It gives insight into how “unsorted” the array is at each step, although it’s not necessary for sorting.

5️⃣ Does the heuristic affect the sorting process here?

- No, it’s just used for display; the sorting is done purely by selection sort.

6️⃣ What are the best and worst cases for Selection Sort?

- Both best and worst cases take O(n²) time because selection sort always scans the unsorted part, even if the array is already sorted.

7️⃣ Can selection sort be optimized using the heuristic?

- Not directly; the heuristic here is for observation, not optimization.

8️⃣ How is the minimum element found in selection sort?

- By scanning the unsorted portion of the array in each iteration.

9️⃣ What is the space complexity of selection sort?

- O(1), because it sorts in place without extra space.

🔟 What other sorting algorithms are more efficient for large datasets?

- Quick Sort, Merge Sort, or Heap Sort, which have O(n log n) average time complexity.

-------------------------------------------------

Prim's Minimum Spanning Tree (MST) Algorithm with Heuristic

-------------------------------------------------

-------------------------------------------------

Code Explanation:

1️⃣ \*\*Importing the sys module\*\*

import sys

The sys module is imported to handle large values like sys.maxsize, which represents the largest integer value, used to initialize key values for comparison.

2️⃣ Graph Class Initialization

class Graph():

def \_\_init\_\_(self, vertices):

self.V = vertices

self.graph = [[0 for column in range(vertices)] for row in range(vertices)]

Graph class: Represents the graph with vertices as the number of nodes.

\_\_init\_\_(self, vertices): Initializes the graph with a 2D list (adjacency matrix) of size vertices x vertices, initialized with 0 (indicating no edges).

3️⃣ Function to Print the MST

def printMST(self, parent, heuristics, total\_cost):

print("Edge \tWeight \tHeuristic Value")

for i in range(1, self.V):

print(f"{parent[i]} - {i} \t {self.graph[i][parent[i]]} \t {heuristics[i]}")

print(f"\nTotal MST Cost: {total\_cost}")

printMST(self, parent, heuristics, total\_cost): Prints the MST, displaying each edge, its weight, and the heuristic value for each vertex.

parent[i]: The parent of vertex i in the MST.

self.graph[i][parent[i]]: The weight of the edge between vertex i and its parent.

heuristics[i]: The heuristic value of vertex i.

4️⃣ Function to Find Minimum Key

python

Copy

Edit

def minKey(self, key, mstSet):

min\_val = sys.maxsize

min\_index = -1

for v in range(self.V):

if key[v] < min\_val and not mstSet[v]:

min\_val = key[v]

min\_index = v

return min\_index

minKey(self, key, mstSet): Finds the vertex with the smallest key value that is not yet included in the MST.

key[v]: The key value of vertex v, representing the minimum edge weight to connect that vertex to the MST.

mstSet[v]: Boolean array, marking whether a vertex is included in the MST.

min\_val: Stores the minimum key value found.

min\_index: Stores the index of the vertex with the minimum key value.

5️⃣ Prim’s Algorithm Implementation

python

Copy

Edit

def primMST(self, heuristics):

key = [sys.maxsize] \* self.V

parent = [None] \* self.V

key[0] = 0

mstSet = [False] \* self.V

parent[0] = -1

total\_cost = 0 # Variable to store total cost of the MST

for cout in range(self.V):

u = self.minKey(key, mstSet)

mstSet[u] = True

for v in range(self.V):

if self.graph[u][v] > 0 and not mstSet[v] and self.graph[u][v] < key[v]:

key[v] = self.graph[u][v]

parent[v] = u

# Now calculate the total cost of the MST by summing the edge weights

total\_cost = sum(self.graph[i][parent[i]] for i in range(1, self.V))

# Print MST and total cost

self.printMST(parent, heuristics, total\_cost)

primMST(self, heuristics): Implements Prim’s algorithm to find the MST:

Initializes the key array with the maximum possible value for each vertex.

Marks the first vertex with key value 0 to start the MST.

Uses minKey() to select the vertex with the minimum key value that hasn't been added to the MST.

For each vertex v, if there is an edge from the selected vertex u to v, and v is not in the MST, the algorithm updates key[v] with the edge weight and assigns u as the parent of v.

Finally, it calculates and prints the MST and its total cost.

6️⃣ Main Program

python

Copy

Edit

if \_\_name\_\_ == '\_\_main\_\_':

print("Lokesh Dhoble 22131")

print("Estimated cost to reach the goal from the current node")

vertices = int(input("Enter the number of vertices: "))

g = Graph(vertices)

print("Enter the graph matrix:")

for i in range(vertices):

row = list(map(int, input().split()))

for j in range(vertices):

g.graph[i][j] = row[j]

# Get heuristic values from the user

heuristics = []

print("Enter the heuristic values for each vertex:")

for i in range(vertices):

h\_value = int(input(f"Heuristic value for vertex {i}: "))

heuristics.append(h\_value)

source\_vertex = int(input("Enter the source vertex: "))

g.primMST(heuristics)

if \_\_name\_\_ == '\_\_main\_\_': Main entry point of the program.

Prompts the user for the number of vertices and the graph matrix (adjacency matrix).

Asks for heuristic values for each vertex.

Calls primMST() to compute the Minimum Spanning Tree.

Explanation of the Algorithm:

1️⃣ Prim’s Algorithm:

It is a greedy algorithm that builds the MST by starting with any arbitrary vertex and adding the closest possible edge to the MST at each step.

2️⃣ Heuristic Values:

The heuristic values represent the estimated cost to reach the goal from the current node. These values are used to guide the selection of the MST edges.

3️⃣ Edge Weights:

The edge weights between nodes represent the cost to travel from one vertex to another. These values are used in Prim's algorithm to determine which edge to add to the MST.

4️⃣ MST Cost:

After the MST is built, the total cost is calculated by summing the weights of all edges included in the MST.

Possible Viva Questions:

1️⃣ What is Prim's Algorithm?

A greedy algorithm that finds the minimum spanning tree (MST) for a connected, undirected graph.

2️⃣ How does Prim's Algorithm differ from Kruskal's Algorithm?

Prim's algorithm grows the MST one vertex at a time, while Kruskal's algorithm sorts all edges first and then adds them one by one to the MST.

3️⃣ What is the role of heuristic values in this implementation?

Heuristic values provide additional information that may guide the algorithm in choosing the next edge based on estimated costs.

4️⃣ How do you find the minimum edge to add to the MST in Prim's algorithm?

The algorithm picks the edge with the smallest weight that connects a vertex in the MST to a vertex outside it.

5️⃣ What is the time complexity of Prim's Algorithm?

The time complexity is O(V^2) when using an adjacency matrix.

6️⃣ Can Prim's algorithm handle disconnected graphs?

No, Prim’s algorithm only works on connected graphs.